# Usando el Ambiente Maude para la Demostración Semi-Automática de Teoremas en Lógica Temporal Lineal de Primer Orden

Patricio Mac Donnell y Nazareno Aguirre
Departamento de Computación
Facultad de Ciencias Exactas, Físico-Químicas y Naturales
Universidad Nacional de Río Cuarto
Ruta 36 Km. 601, Río Cuarto (5800),
Córdoba, Argentina
{patricio,naguirre}@dc.exa.unrc.edu.ar

**Resumen**

En este artículo, presentamos una implementación modular de un demostrador semi-automático de teoremas para la lógica temporal lineal de primer orden, realizada en el ambiente Maude. Maude es un ambiente de programación y especificación, basado en la lógica de reescritura, propuesta por J. Meseguer. Entre las aplicaciones de la lógica de reescritura, se destaca la aplicación de ésta como ambiente para la implementación de otras lógicas. Aprovechamos en este artículo esta singular característica de la lógica de reescritura, y la simplicidad, expresividad y eficiencia de Maude, para construir un demostrador de teoremas interactivo, para la lógica temporal lineal de primer orden, basado en un cálculo de secuentes para ésta.

## 1 Introducción

La creciente aplicación de la computación en sistemas críticos [7], es decir, sistemas cuyo mal funcionamiento puede causar daños económicos importantes o la pérdida de vidas humanas, hace necesaria la aplicación de métodos rigurosos de desarrollo de software. Los métodos formales de desarrollo están sustentados por sólidas bases matemáticas, y con frecuencia involucran lógicas de algún tipo. Por esto, su utilización requiere en general la manipulación de fórmulas en la lógica correspondiente al formalismo utilizado, y la deducción en ésta.

Es sabido que, en la práctica, la aplicación de métodos formales requiere la manipulación de expresiones extensas, y las demostraciones de corrección, o de que alguna porción de código posee cierta propiedad, son en general largas. Por esto, para la aplicación exitosa de los métodos formales, es esencial contar con herramientas de software, que permitan automatizar, o asistir, en las actividades del especificador o desarrollador.

Las lógicas modales, y en particular las lógicas temporales, han tenido gran éxito en la especificación y verificación formal de sistemas reactivos, es decir, aquellos sistemas que mantienen una interacción con el ambiente. Parte de este éxito se debe a que algunas técnicas de verificación asociadas a estas lógicas son completamente automatizables; el caso más conocido es el de *model checking* [3]. Sin embargo, algunos investigadores han advertido que las variantes más utilizadas de las lógicas

modales, que son proposicionales, poseen serias limitaciones en poder expresivo, y por lo tanto es importante considerar sus extensiones de primer orden [4]. Una de estas extensiones es la denominada *lógica de Manna-Pnueli*, una lógica temporal lineal de primer orden. Esta lógica es el lenguaje de especificaciones de un ambiente de verificación conocido como STeP [2]. STeP combina model checking con métodos deductivos para la verificación de propiedades temporales de sistemas reactivos. Más precisamente, la herramienta combina model checking simbólico con un cálculo de secuentes, para realizar demostraciones de propiedades de sistemas. Más aún, algunas técnicas de verificación soportadas por la herramienta STeP permiten aprovechar las características del sistema reactivo objeto de la demostración para realizar pruebas de propiedades (por ejemplo, demostrando que todas las transiciones preservan una propiedad para demostrar su invarianza, etc).

Si bien STeP es una herramienta muy potente, el proyecto STeP parece estar en decaimiento en la actualidad, y por razones de licencia, no se puede acceder al código de la herramienta para realizar modificaciones o extensiones. Parcialmente motivados por esto, hemos trabajado en el desarrollo de un prototipo de un demostrador semi-automático de teoremas para la lógica de Manna-Pnueli. La herramienta elegida para esto es Maude [1]. Maude es un ambiente de programación y especificación, basado en la lógica de reescritura, propuesta por J. Meseguer [6]. Entre las aplicaciones de la lógica de reescritura, se destaca la aplicación de ésta como ambiente para la implementación de otras lógicas. En este artículo presentamos la caracterización de la lógica de Manna-Pnueli en Maude, y mostramos cómo esta caracterización puede aprovecharse para utilizar el entorno Maude para realizar demostraciones interactivas de teoremas de esta lógica. Presentamos la caracterización modularmente, las características de la especificación, y ejemplos de uso de este prototipo.

# 2 La Lógica de Reescritura

Una *signatura*, en lógica de reescritura, es una teoría ecuacional $(\Sigma, E)$, donde $\Sigma$ es una signatura y $E$ es un conjunto de $\Sigma$-ecuaciones (la lógica de reescritura esta parametrizada por la lógica subyacente que utiliza; ésta puede ser no tipada, tipada, orden-tipada o la recientemente desarrollada logica ecuacional con ecuaciones de pertenencia a tipos).

Una *teoría de reescritura* $\mathcal{R}$, es una generalización de una teoría ecuacional $(\Sigma, E)$, a la cual se le agregan reglas de reescritura que tienen la forma

$$[t]_E \longrightarrow [t']_E$$

donde t y t' son $\Sigma$-términos que pueden contener variables y $[t]_E$ denota la clase de equivalencia del término t, módulo las ecuaciones de $E$. Es decir que para escribir las reglas de reescritura debemos utilizar los términos canónicos que representan a cada clase de equivalencia.

Dada una teoría $\mathcal{R}$, decimos que $\mathcal{R}$ vincula una sentencia $[t]_E \longrightarrow [t']_E$ y escribimos $\mathcal{R} \vdash [t]_E \longrightarrow [t']_E$, si y sólo si $[t]_E \longrightarrow [t']_E$ puede ser obtenida de alguna de las siguientes reglas de deducción (asumiendo que los términos estan bien formados y $t(\overline{w}/\overline{x})$ denota la substitucion simultánea de $w_i$ por $x_i$ en $t$):

(i) **Reflexividad.** Para cada $[t] \in T_{\Sigma,E}(X)$, $\quad \dfrac{}{[t] \longrightarrow [t]}$

(ii) **Congruencia.** Para cada $f \in \Sigma_n, \quad n \in \mathbb{N}$,

$$\frac{[t_1] \longrightarrow [t'_1] \quad \ldots \quad [t_n] \longrightarrow [t'_n]}{[f(t_1, ..., t_n)] \longrightarrow [f(t'_1, ..., t'_n)]}$$

(iii) **Reemplazo.** Para cada regla $r : [t(x_1, ..., x_n)] \longrightarrow [t'(x_1, ..., x_n)] \in \mathcal{R}$,

$$\frac{[w_1] \longrightarrow [w'_1] \quad \ldots \quad [w_n] \longrightarrow [w'_n]}{t(\overline{w}/\overline{x}) \longrightarrow t'(\overline{w'}/\overline{x})}$$

(iv) **Transitividad.**

$$\frac{[t_1] \longrightarrow [t_2] \qquad [t_2] \longrightarrow [t_3]}{[t_1] \longrightarrow [t_3]}$$

La lógica de reescritura es muy útil para especificar *sistemas concurrentes*, los cuales se representan con estados y transiciones entre ellos. La signatura de una teoría de reescritura describe la estructura de los estados de un sistema (por ej: árboles, conjuntos, etc), en tanto que las reglas de reescritura de la teoría describen las transiciones del mismo. Cada paso de reescritura es una transición en un sistema paralelo/concurrente.

Alternativamente, podemos adoptar un punto de vista lógico; en este caso la signatura representa la forma en que se escriben las fórmulas de la lógica objeto y las reglas de reescritura representan las metareglas de deducción en dicha lógica.

# 3 La Lógica Temporal Lineal de Primer Orden

Podemos ver a la lógica temporal lineal de primer orden como una extensión de la lógica de primer orden a la cual se le agregan los operadores temporales $\Box$, $\bigcirc$ , $\mathcal{U}$ y a partir de ellos podemos obtener otros como $\Diamond$ y $\mathcal{W}$. El significado intuitivo de los mismos es el siguiente:

- $\Box\alpha$: siempre en el futuro sucede $\alpha$.
- $\bigcirc\alpha$: en el siguiente estado del tiempo sucede $\alpha$.
- $\alpha\mathcal{U}\beta$: $\alpha$ ocurre hasta que ocurre $\beta$.
- $\Diamond\alpha$: en algún instante futuro sucede $\alpha$.
- $\alpha\mathcal{W}\beta$: $\alpha$ ocurre hasta que ocurre $\beta$, o siempre ocurre $\alpha$.

Teniendo en cuenta que un programa en ejecución puede interpretarse como una secuencia de estados (es decir, una traza de ejecución es una secuencia potencialmente infinita de estados), estos nuevos operadores nos permiten especificar las propiedades que debrían tener tales ejecuciones. Por ejemplo, si deseamos saber si cierta aserción $\alpha$ es un invariante de un sistema dado, deberemos verificar que todas las trazas de ejecución del mismo satisfagan la fórmula temporal:

$$\Box\alpha$$

Si, en cambio, deseamos comprobar si un sistema dado posee la propiedad de que, la ocurrencia de un evento $\alpha$ (representado por una fórmula) produce la ejecución eventual de un evento o acción $\beta$ (representado, nuevamente, por una fórmula), deberemos verificar que todas las trazas de ejecución del sistema satisfagan la fórmula temporal:

$$\Box(\alpha \rightarrow \Diamond\beta)$$

La lógica temporal lineal de primer orden nos permite especificar eventos o acciones ($\alpha$ y $\beta$ en los ejemplos anteriores) expresables no sólo mediante proposiciones simples, sino también utilizando cuantificación de variables individuales. Por ejemplo, supongamos que tenemos un sistema que, entre otras cosas, consta de una

variable entera de programa $v$. Si queremos especificar el hecho de que, en toda ejecución posible del sistema, $v$ toma todos los valores enteros posibles, podemos hacerlo de la siguiente manera:

$$\forall x \in \mathsf{int} : \Diamond(v = x)$$

Por razones de espacio, nos es imposible describir en detalle esta lógica. El lector interesado puede consultar [4].

# 4 El Ambiente Maude

*Maude* es un lenguaje de programación declarativa basado en lógica de reescritura. Todos los conceptos hasta aquí abordados estan soportados por la versión 2.0 de *Maude*.

## 4.1 Módulos

En maude, los módulos definen un álgebra, es decir definen un conjunto de *tipos* y *operaciones sobre ellos*. Los modulos principales son los *módulos funcionales* y los *módulos de sistema* Cada módulo es declarado con las siguientes palabras claves:

```
fmod NAME is ... endfm
mod NAME is ... endm
```

Al igual que en muchos lenguajes de programación, podemos importar un módulo desde otro:

```
protecting NOMBREMODULO.
including NOMBREMODULO.
extending NOMBREMODULO.
```

La palabra clave *protecting* nos permite importar sin modificar el modulo, *including* permite importar y cambiar el significado de las operaciones pero no su interfaz y por último *extending* nos permite importar y agregar sorts y nuevas operaciones al módulo importado.

## 4.2 Sorts y Variables

Un *sort* define un tipo de valores. Los nombres de *sorts* no pueden tener espacios en blanco ni '{', '[', '(' a menos que sean precedidos por un apóstrofe. Un sort se declara en un modulo con la palabra clave `sort` y un punto al final.

```
sort integer .
sorts integer decimal .
```

También podemos declarar subsorts:

```
subsort integer < decimal .
```

Una *variable* es un valor indefinido para un sort. Las variables se declaran con la palabra `var` o `vars`.

```
var x : number .
vars c1 c2 c3 : color .
```

## 4.3 Operaciones

Una operación se declara con la palabra `op`, el nombre de la operación, dos puntos, nombres de sorts correspondientes a los argumentos, una flecha (`->`), el sort del resultado y finalmente un punto. Cabe acotar que las operaciones pueden ser subfijas, infijas o prefijas, para hacer notar esto, se utilizan guiones que indican la posición de los argumentos.

```
op _+_ : Nat Nat -> Nat .
op +(_ _) : Nat Nat -> Nat .
op (_ _)+ : Nat Nat -> Nat .
```

## 4.4 Constructores y Atributos de Operación

En toda álgebra hay operaciones básicas, llamadas contructores, que son las que definen la estructura básica de la misma. Para designar a una operación como constructora, agregamos "`[ctor]`" despues del sort del resultado y antes del punto.

```
fmod Nat is
   sort Nat .
   op 0 : -> Nat [ctor] .
   op suc_ : Nat -> Nat [ctor] .
endfm
```

Hay otros atributos que pueden ser agregados a las operaciones, que nos permiten "ahorrarnos" la escritura de ciertas ecuaciones, ellos son: `assoc comm id:oper`. Los mismos indican que la operación a la cual están vinculados es asociativa, conmutativa y tiene como elemento neutro al operador `oper`, respectivamente.

## 4.5 Módulos Funcionales

Los módulos funcionales contienen definiciones de sorts, definiciones de operaciones y ecuaciones, ellos no contienen reglas de reescritura. Es decir describen la estructura estática del problema, pero no los cambios de estado del mismo, lo cual se define en los módulos de sistema a través de reglas de reescritura.

Las ecuaciones sirven para proveer a *Maude* de ciertas reglas para simplificar una expresión a su forma mas simple, también llamada *forma canónica*, la cual es una expresión equivalente en la que los operadores son todos contructores. Ellas se declaran con la palabra `eq`. Consideremos, por ejemplo, la siguiente especificación y la utilización de ecuaciones en ella:

```
fmod Nat is
   sort Nat .
   op 0 : -> Nat [ctor].
   op suc : Nat -> Nat [ctor] .
   op _ + _ : Nat Nat -> Nat .
   vars M N : Nat .
   eq 0 + N = M .
   eq suc(M) + N = suc(M + N)
endfm
```

Las ecuaciones condicionales son declaradas con la palabra `ceq`, la ecuación correspondiente, la palabra `if`, la condición y finalmente un punto.

```
op diferentes?(_ , _) : Nat Nat -> Bool .
ceq diferentes?(N , M)=true if N =/= M .
```

Las ecuaciones de pertenencia a tipo son declaradas con la palabra `me` y las ecuaciones de pertenencia a tipo con condiciones son declaradas con la palabra `cmb`. Considere la siguiente especificación y la utilización de ecuaciones más sofisticadas en ella:

```
fmod Nat32 is
   protecting Nat .
   sort Nat32 < Nat .
   vars N M : Nat .
   cmb N : Nat32 if N < 32 .
   mb M : Nat32 .
endfm
```

### 4.6 Módulos de Sistema

Como decíamos en la sección anterior, las reglas de rescritura nos permiten especificar transiciones entre estados representados por módulos funcionales. La gran diferencia entre las ecuaciones y las reglas de reescritura es que las primeras solo sirven para reducir un término a su equivalente canónico. La característica principal de las reglas de reescritura es su *irreversibilidad*. Una transición va de un estado hacia otro y no es posible la reversa si no es a través de otra regla. Una regla de rescritura se declara con la palabra clave `rl`, en nombre entre corchetes, dos puntos, estado origen, una "=>", estado de llegada y finalmente un punto. Ejemplo:

```
mod Clima is
   sort condicionclimatica .
   op diasoleado : -> condicionclimatica .
   op dialluvioso : -> condicionclimatica .
   rl [soleadolluvioso] : diasoleado => dialluvioso .
endm
```

## 5 Demostrador Semi-Automático de Teoremas para LTL de Primer Orden en Maude

Este demostrador consta de dos partes fundamentales:

- La primera de ellas se vale de los *módulos funcionales* de maude para definir las operaciones que nos permitirán escribir las fórmulas y secuentes de la lógica temporal lineal de primer orden.
- la segunda utiliza los *módulos de sistema* de maude para describir, a través de reglas de reescritura, las reglas del cálculo de secuentes para dicha lógica.

La idea principal de este demostrador es partir del secuente que queremos demostrar, y utilizar las reglas de reescritura para obtener nuevos secuentes que pasarán a ser nuestros objetivos a demostrar. La demostración termina cuando todos los subobjetivos han sido reducidos a `empty`. Para lograr esto, el usuario interactúa con el demostrador a través de los siguientes comandos:

- load: ingresa el secuente a demostrar.
- apRule: Toma como parámetro el nombre de una regla de reescritura, una lista de substituciones de variables, en dicha regla, por términos y la aplica.

- prop: intenta llegar al estado *empty*, a partir del secuente actual, aplicando reglas de la logica proposicional.

- checkLoop: reduce a *empty* un secuente si hemos llegado a otro idéntico, aplicando al menos una vez la regla *next* y siempre que en algún paso de la deducción haya un secuente que contenga una fórmula insatisfecha (*unfulfilled formula* en el sentido de [5])

## 5.1 Signaturas

En esta sección mostraremos las operaciones y ecuaciones más importantes de las signaturas que describen las diferentes lógicas. El módulo `LOG_LIN` define los símbolos normalmente utilizados en la lógica de primer orden. El mismo importa el módulo `ADT` el cuál contiene la definición de las fórmulas atómicas; éstas se definen en tiempo de ejecución partir de las definiciones de variables, constantes y predicados que son suministradas por el usuario en un archivo de declaraciones.

```
fmod LOG_LIN is

pr ADT .

sorts Formula .
subsorts Atom < Formula .

ops true false : -> Atom [ctor] .
op  !_    : Formula -> Formula [ctor prec 23] .
op _&_   : Formula Formula -> Formula
[prec 55 gather (E e) comm] .
op _|_   : Formula Formula -> Formula
[ctor prec 59 gather (E e) comm] .
op _->_  : Formula Formula -> Formula [prec 65] .
op _<->_ : Formula Formula -> Formula [prec 65] .

vars A B : Formula .

eq A & B = ! (! A | ! B) .
eq A -> B = ! A | B .
eq A <-> B = (A -> B) &  (B -> A) .
eq ! true = false .
eq ! false = true .
eq A | true = true .
eq A & false = false .
eq A | false = A .
eq A & true = A .
eq A | ! A = true .
eq A & ! A = false .
eq ! ! A = A .
endfm
```

El módulo `LOG_PRIM_ORD` extiende al anterior para dar las reglas de formación de fórmulas de la lógica de primer orden.

```
fmod LOG_PRIM_ORD is
```

```
ext LOG_LIN .

*** Cuantificador existencial.
op \/_._ : Var Formula -> Formula [ctor prec 22] .

*** Cuantificador universal .
op /\_._ : Var Formula -> Formula [ctor prec 21].

vars A  : Formula .
vars X  : Var .

eq ! /\ X . A = \/ X . (! A) .
eq ! \/ X . A = /\ X . (! A) .

endfm
```

El módulo `LTL` extiende al módulo `LOG_PRIM_ORD` para definir las formulas de la lógica temporal lineal de primer orden.

```
fmod LTL is

ext LOG_PRIM_ORD .

op []_ : Formula -> Formula [ctor prec 23] .
op O_ : Formula -> Formula [ctor prec 23] .
op _U_ : Formula Formula -> Formula [ctor prec 45] .
op <>_ : Formula -> Formula [ctor prec 23] .
op _W_ : Formula Formula -> Formula [ctor prec 45] .

vars A B : Formula .
vars T1 T2 : Term .

eq ! <> (A) =  [] (! A) .
eq ! [] (A) = <> (! A) .
eq (A) U (B) |  A  = (A) W (B) .
eq ! (! ((A) W (B)) | [] (! A)) = (A) U (B) .

endfm
```

Por último contamos con el módulo `SEC` en el cuál definimos la estructura de un secuente y también la operacion `op _ _ : SecSet SecSet -> SecSet` sobre la que actuarán las reglas de reescritura para describir las reglas de transformación de secuentes. Cabe destacar que la operación *load* carga el secuente en un SecSet y las reglas de reescritura se van aplicando siempre al primer elemento de éste. Cada vez que un secuente es reducido a empty la ecuación `eq empty S = S` pone en primer lugar al siguiente secuente.

```
fmod SEC is

pr LTL .

*****************************SECUENTES*****************************

sorts FormSet Secuente .
```

```
subsort Formula < FormSet .

op _,_ : FormSet FormSet -> FormSet [ctor comm assoc prec 70] .
op _|-_ : FormSet FormSet -> Secuente [ctor prec 75] .

**********************CALCULO DE SECUENTES*************************

sorts SecSet .
subsort Secuente < SecSet .

  op empty : -> SecSet .
op _ _  : SecSet SecSet -> SecSet  [ctor assoc comm prec 80] .

var S : SecSet .

eq empty S = S .

endfm
```

## 5.2 Razonamiento en Lógica de Primer Orden

En esta sección describiremos las reglas de rescritura que definen el cálculo de secuentes de la lógica de primer orden. Las reglas que utilizamos, excepto las dos primeras, son derivadas del cálculo de secuentes de lado derecho, es decir, secuentes que tienen la forma: `|- A,c,d`, esto es posible debido a que por medio de las reglas ,`conv` y `conv1` podemos transformar cualquier secuente en otro en el cual el lado izquierdo es vaciío.

```
mod CAL_SEC is

inc SEC .

vars A B C : FormSet .
vars c d : Formula .
vars X Y : Var .
var T : Termino .

  *** Reglas de transformacion.

rl [conv] :   B , c |- A
          => ---------------
              B |- A , ! c .

rl [conv1] :  c |- A
           => ------------------
              true |- A , ! c .

*** Reglas para Logica Proposicional.

rl [id] :  B |- c , ! c
        => -------------
              empty .
```

```
rl [cut] :           B |- A , C
         => ------------------------------
             B |- A , c    B |- C , ! c .

rl [wk] :  B |- A , c
        => ------------
           B |- A .

rl [dis] :  B |- A , c | d
         => -----------------
            B |- A , c , d .

rl [con] :          B |- A , C , ! (c | d)
         => -----------------------------------
             B |- A , ! c       B |- C , ! d .

rl [t] :  B |- true
       => -----------
           empty .

  *** Reglas para la Logica de Primer Orden.

rl [univ] :  B |- A , /\ X . c
          => ------------------
             B |- A , c .

rl [exist] :  B |- A , \/ X . c
           => ---------------------
              B |- A , c [T / X] .
```

## 5.3 Operadores Temporales

Las reglas aquí descriptas también forman parte del módulo `CAL_SEC` y son las que tratan los operadores temporales.

```
rl [always] :             B |- A , [] (c)
            => ----------------------------------------
               B |- A , c        B |- A , O [] (c) , ! c .


rl [possibly] :  B |- A , <> (c)
              => ------------------------
                 B |- A , c , O <> (c) .


rl [until] :                    B |- A , (c) U (d)
           =>-------------------------------------------------
               B |- A , c , d       B |- A , O ((c) U (d)) , d .


rl [wuntil] :                   B |- A , (c) W (d)
            => ---------------------------------------------------
```

```
                B |- A , c , d     B |- A , O ((c) W (d)) , d , ! c .


rl [next] :  B |- A , O (c) , ! O (d)
          => ------------------------
             B |- A , c , ! d .
```

# 6 Un Ejemplo

En esta sección mostramos un breve ejemplo de utilización del demostrador interactivo. Debido a restricciones de espacio, nos es imposible mostrar la demostración de un teorema complejo. Demostramos simplemente lo siguiente:

$$\bigcirc(\forall x : p(x)) \vdash \bigcirc p(f(c))$$

```
*** Cargamos el Secuente.
Maude> (load O /\ v1 . p(v1) |- O p(f(c3)))
rewrites: 35 in 10ms cpu (10ms real) (3500 rewrites/second)
O /\ v1 . p(v1)|- O p(f(c3))

*** Aplicamos la regla conv1, sin substituciones.
Maude> (apRule conv1 no)
rewrites: 62 in 10ms cpu (10ms real) (6200 rewrites/second)
true |- ! O /\ v1 . p(v1) , O p(f(c3))

*** Aplicamos la regla conv1, sin substituciones.
Maude> (apRule conv1 no)
rewrites: 142 in 20ms cpu (20ms real) (7100 rewrites/second)
true |- false , ! O /\ v1 . p(v1) , O p(f(c3))

*** Aplicamos la regla next, substituyendo la
*** variable c de la regla por p(f(c3)).
Maude> (apRule next c <- p(f(c3)))
rewrites: 217 in 10ms cpu (10ms real) (21700 rewrites/second)
true |- false , p(f(c3)) , \/ v1 .(! p(v1))

*** Aplicamos la regla exist, substituyendo la variable T por p(f(c3)).
Maude> (apRule exist T <- f(c3))
rewrites: 211 in 20ms cpu (20ms real) (10550 rewrites/second)
true |- false , p(f(c3)) , ! p(f(c3))

*** Aplicamos la regla wk, substituyendo la variable c por false.
Maude> (apRule wk c <- false)
rewrites: 133 in 10ms cpu (10ms real) (13300 rewrites/second)
true |- p(f(c3)) , ! p(f(c3))

*** Aplicamos la regla id, sin substituciones.
Maude> (apRule id no)
rewrites: 41 in 10ms cpu (10ms real) (4100 rewrites/second)
**************************Empty*********************************
```

# 7 Conclusiones

Hemos presentado una implementación de un demostrador semi-automático de teoremas para una lógica temporal lineal de primer orden en el ambiente de reescritura Maude. Para realizar este trabajo, hemos tenido que codificar la lógica de Manna-Pnueli, objeto de nuestro estudio, en la lógica de reescritura. La codificación de la misma ha sido relativamente directa. Nos basamos para esto en un cálculo de secuentes para la lógica de Manna-Pnueli.

La implementación del demostrador fue realizada de manera modular. La implementación de reglas "interactivas" (reglas en las cuales es indispensable la intervención del usuario, como en la regla de cut), fue necesario utilizar el módulo `loop` de Maude. Algunas complicaciones fueron encontradas al momento de extender la implementación del cálculo para lógica de primer orden con operadores temporales. La dificultad principal fue en la implementación de la regla `checkLoop`, la cual corresponde a una especie de regla de inducción de la lógica de Manna-Pnueli (esencialmente, corresponde al axioma LTL $(\alpha \rightarrow \bigcirc\alpha) \rightarrow (\alpha \rightarrow \Box\alpha)$).

Como mencionamos anteriormente, este trabajo está parcialmente motivado en el aparente decaimiento del proyecto STeP, y la imposibilidad de acceder al código de esta herramienta por razones de licencia. El demostrador construido es, actualmente, sólo un prototipo, y múltiples estudios de performance restan por hacerse para compararlo con STeP. No es nuestro objetivo, de ninguna manera, reemplazar a STeP, una herramienta muy poderosa, que combina inteligentemente métodos automáticos de verificación (model checking), técnicas de interpretación abstracta y deducción (semi-)automática. La eficiencia de Maude indicaría, en principio, que en el futuro podremos conseguir un demostrador poderoso y eficiente para la lógica estudiada.

# Referencias


[1] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer and C. Talcott, *The Maude 2.0 System*, In Proceedings of Rewriting Techniques and Applications, 2003, Springer-Verlag LNCS 2706, 76-87, June 2003.

[2] N. Bjorner, A. Browne, M. Colon, B. Finkbeiner, Z. Manna, H. Sipma and T. Uribe, *Verifying Temporal Properties of Reactive Systems: a STeP Tutorial*, in Formal Methods in System Design, vol 16, 2000

[3] E. Clarke, O. Grumberg y D. Peled, *Model Checking*, MIT Press, 1999.

[4] Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems*, Springer-Verlag, 1991.

[5] Z. Manna and A. Pnueli, *Temporal Verification of Reactive Systems - Safety -*, Springer, 1995.

[6] N. Martí-Oliet y J. Meseguer, *Rewriting Logic as a Logical and Semantic Framework*, Technical Report SRI-CSL-93-05, SRI International, Computer Science Laboratory, August 1993.

[7] N. Storey, *Safety-Critical Computer Systems*, Addison-Wesley, 1996.